ELEC 8590
Physical Design Automation for VLSI and FPGAs

Lecture 2:
Definitions of PD Tasks, Review of Algorithms, Complexity Analysis & Data Structures

Mohammed Khalid

Department of Electrical and Computer Engineering
University of Windsor

References and Copyright

- Slide sources (including notes):
  - Prof. Kia Bazargan, University of Minnesota
  - Prof. Rajesh Gupta, University of California, Irvine
  - Dr. Naveed Sherwani (Companion slides with textbook)
  - Prof. Scott Hauck, University of Washington
  - Prof. Jonathan Rose, University of Toronto
  - Dr. Habib Youssef, Tunisia

Definitions of Physical Design Tasks

- See PDF file "590_lec2_partial"

Optimization

- Each of the steps in PD flow involve choosing the best among different available choices => optimization
  - this is a key point in CAD
- Optimization means to minimize or maximize a function of many variables subject to certain constraints
  - the function is called objective function
  - combinatorial optimization implies the variables are required to belong to a discrete set, typically a subset of integers.
- We will study different optimization algorithms used for solving problems in physical design automation.

Algorithm

- An algorithm defines a procedure for solving a computational problem
  - Examples:
    - Quick sort, bubble sort, insertion sort, heap sort
    - Dynamic programming method for the knapsack problem
- Definition of complexity
  - Run time on deterministic, sequential machines
  - Based on resources needed to implement the algorithm
    - Needs a cost model: memory, hardware/gates, communication bandwidth, etc.
    - Example: RAM model with single processor

      ➔ running time $\propto$ # operations

[©Gupta]

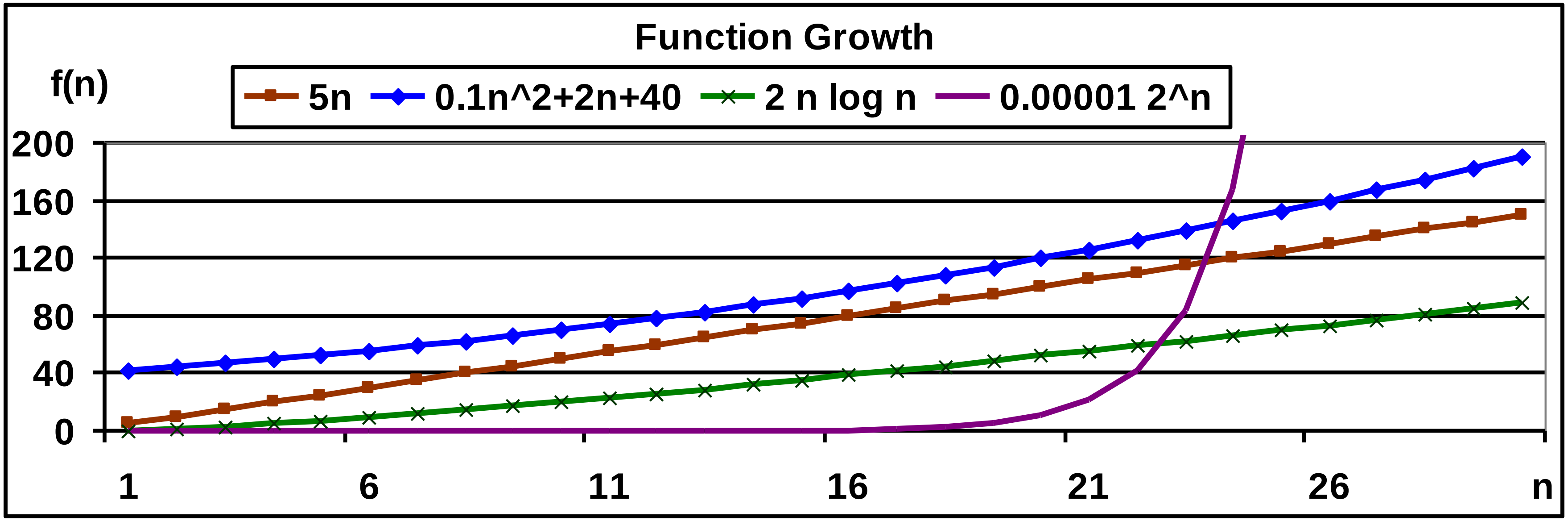
Algorithm (cont.)

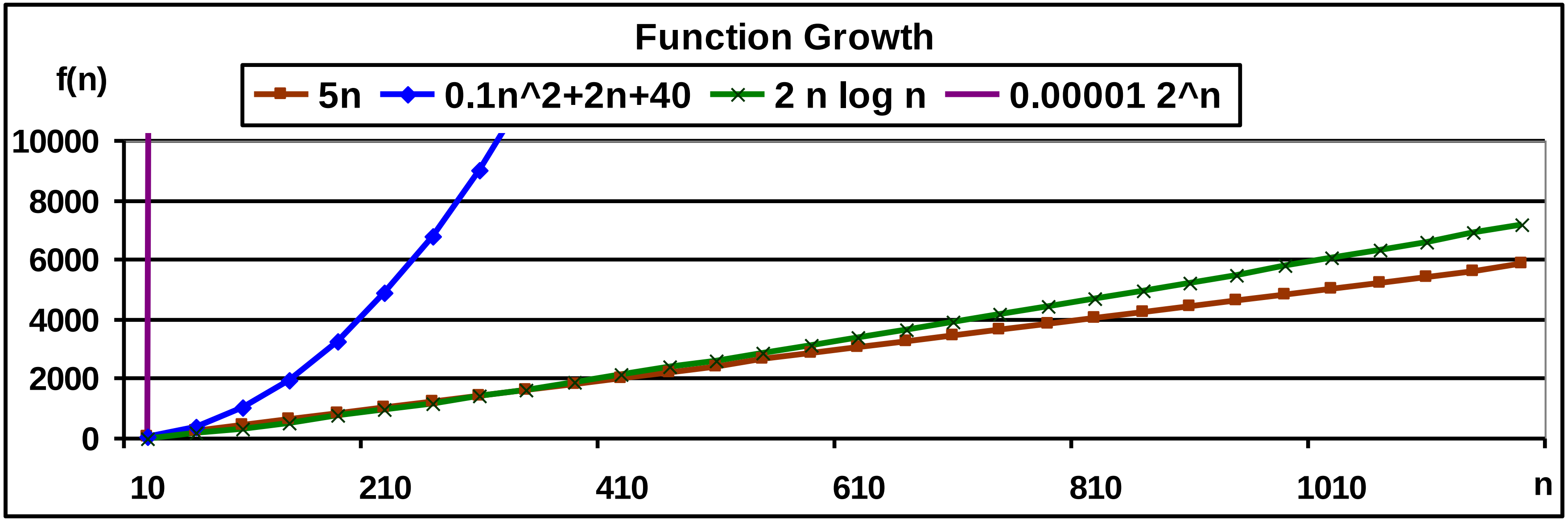
- Definition of complexity (cont.)
  - Example: Bubble Sort →
  - Scalability with respect to input size is important
    - How does the running time of an algorithm change when the input size doubles?
    - Function of input size (n). Examples: n^2+3n, 2^n, n log n, ...
    - Generally, large input sizes are of interest (n > 1,000 or even n > 1,000,000)
    - What if I use a better compiler? What if I run the algorithm on a machine that is 10x faster?

```
for (j=1 ; j< N; j++) {
   for (i=j; i < N-1; i++) {
       if (a[i] > a[i+1]) {
         hold = a[i];
         a[i] = a[i+1];
         a[i+1] = hold;
       }
     }
   }
```

[©Gupta]

Function Growth Examples





Asymptotic Notions

- Idea:
  - A notion that ignores the "constants" and describes the "trend" of a function for large values of the input
- Definition
  - Big-Oh notation $\quad f(n) = O\ (\ g(n)\ )$
    if constants K and n_0 can be found such that:
    $\forall\ n \geq n_0,\ f(n) \leq K.\ g(n)$

    g is called an "upper bound" for f
    (f is "of order" g: f will not grow larger than g by more than a constant factor)

    Examples: $\quad 1/3\ n^2 = O\ (n^2)$
    $\quad 0.02\ n^2 + 127\ n + 1923 = O\ (n^2)$

Asymptotic Notions (cont.)

- Definition (cont.)
  - Big-Omega notation $f(n) = \Omega (g(n))$
    if constants K and n_0 can be found such that:
    $\forall\ n \geq n_0,\ f(n) \geq K.\ g(n)$

    g is called a "lower bound" for f

  - Big-Theta notation $f(n) = \Theta (g(n))$
    if g is both an upper and lower bound for f
    Describes the growth of a function more accurately than O or Ω
    Example:
    $$n^3 + 4\ n \neq \Theta\ (n^2)$$
    $$4\ n^2 + 1024 = \Theta\ (n^2)$$

Asymptotic Notions (cont.)

- How to find the order of a function?
  - Not always easy, esp if you start from an algorithm
  - Focus on the "dominant" term
    - $4\ n^3 + 100\ n^2 + \log n \quad \rightarrow \quad O(n^3)$
    - $n + n \log(n) \quad \rightarrow \quad n \log(n)$
  - $n! = K^n \quad > \quad n^K \quad > \quad \log n \quad > \log \log n \quad > \quad K$

    $\Rightarrow n > \log n, \quad n \log n > n, \quad n! > n^{10}.$
- What do asymptotic notations mean in practice?
  - If algorithm A has "time complexity" $O(n^2)$ and algorithm B has time complexity $O(n \log n)$, then algorithm B is better
  - If problem P has a lower bound of $\Omega(n \log n)$, then there is NO WAY you can find an algorithm that solves the problem in $O(n)$ time.

Problem Tractability

- Problems are classified into “easier” and “harder” categories
  - Class P: a polynomial time algorithm is known for the problem (hence, it is a tractable problem)
  - Class NP (non-deterministic polynomial time):
    ~ polynomial solution not found yet
    (probably does not exist)
    ➔ exact (optimal) solution can be found using an algorithm with exponential time complexity
- Unfortunately, most CAD problems are NP
  - Be happy with a “reasonably good” solution
  - Exact solutions are possible but they will take many years to compute, even for small input sizes! e.g. $O(n!)$ or $O(2^n)$

[©Gupta]

Algorithm Types

- Based on quality of solution and computational effort
  - Deterministic
  - Probabilistic or randomized
  - Approximation
  - Heuristics: local search

[©Gupta]

Deterministic Algorithm Types

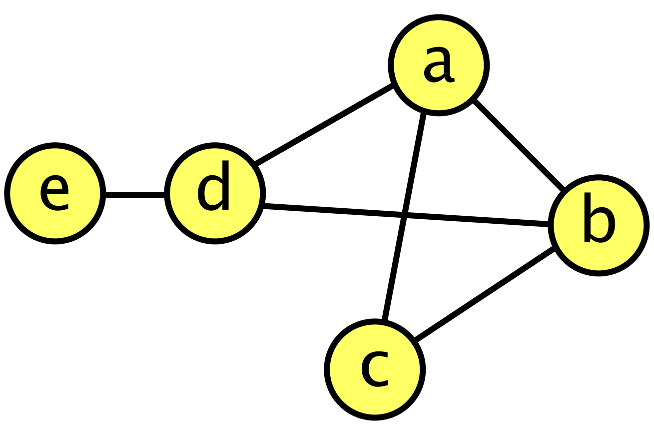
- Algorithms usually used for P problems
  - Exhaustive search! (aka exponential)
  - Dynamic programming
  - Divide & Conquer (aka hierarchical)
  - Greedy
  - Mathematical programming
  - Branch and bound
- Algorithms usually used for NP problems (not seeking "optimal solution", but a "good" one)
  - Greedy (aka heuristic)
  - Genetic algorithms
  - Simulated annealing
  - Restrict the problem to a special case that is in P

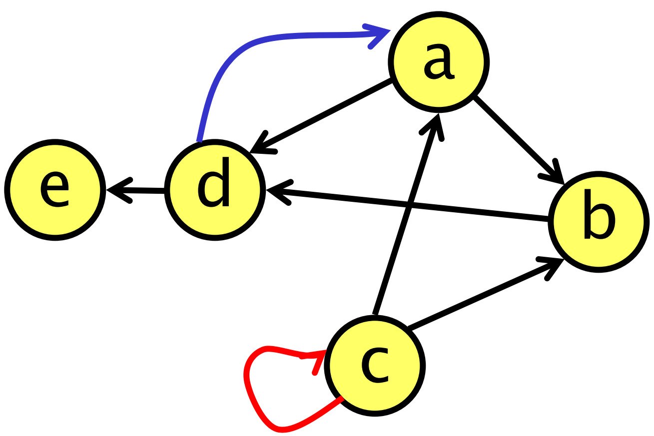
Data Structures

- Review basic data structures
  - arrays, linked lists, stacks and queues
- More advanced data structures
  - priority queues, search trees, graphs
- Important programming tip: Use object oriented programming (C++ or Java) for CAD tool development, you will save hundreds of hours in SW development and testing.
  - Well tested class libraries available for basic and advanced data structures
  - Do not reinvent the wheel!
  - Real world CAD: 90% effort on SW development and 10% on algorithm development - we will do more algorithms and relatively less SW development.

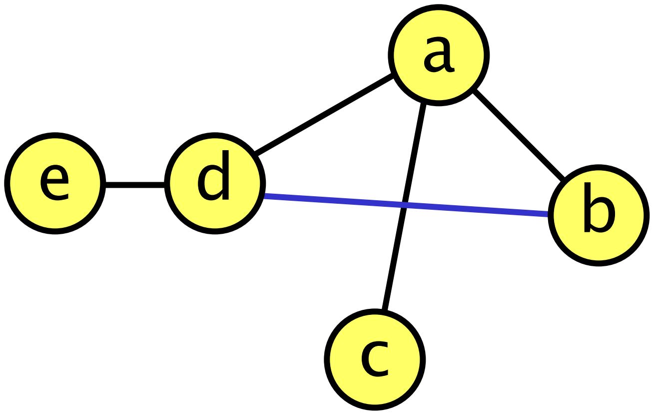
Graph Definition

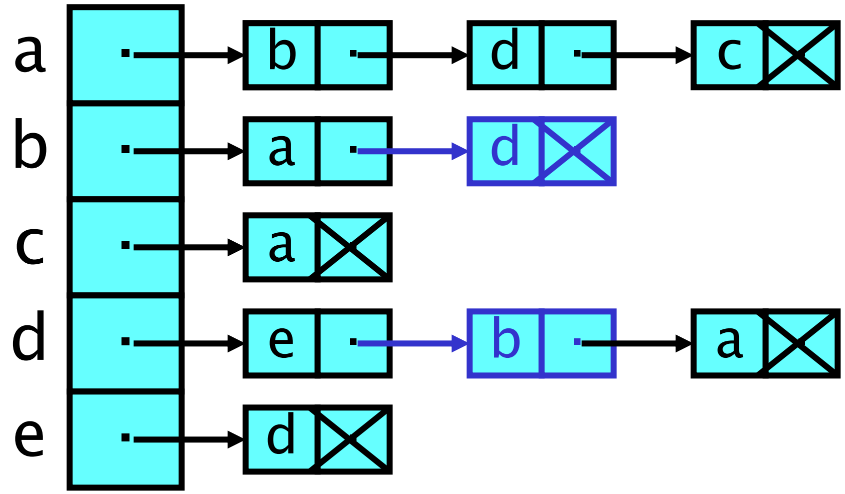
- Graph: set of "objects" and their "connections"
- Formal definition:
  - $G = (V, E)$, $V=\{v_1, v_2, \ldots, v_n\}$, $E=\{e_1, e_2, \ldots, e_m\}$
  - V: set of vertices (nodes), E: set of edges (links, arcs)
  - Directed graph: $e_k = (v_i, v_j)$
  - Undirected graph: $e_k=\{v_i, v_j\}$
  - Weighted graph: $w(e_k)$ is the "weight" of e_k.

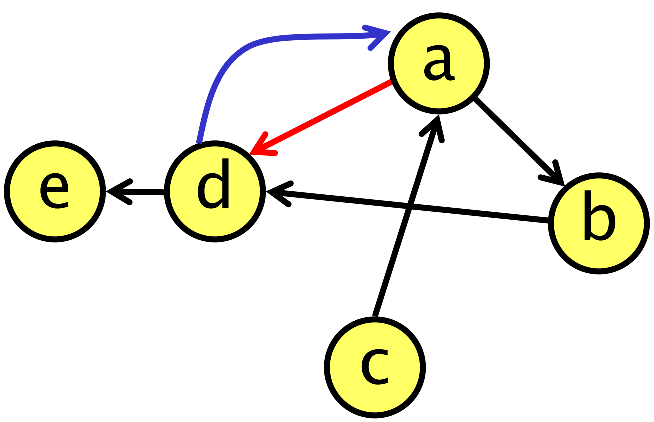


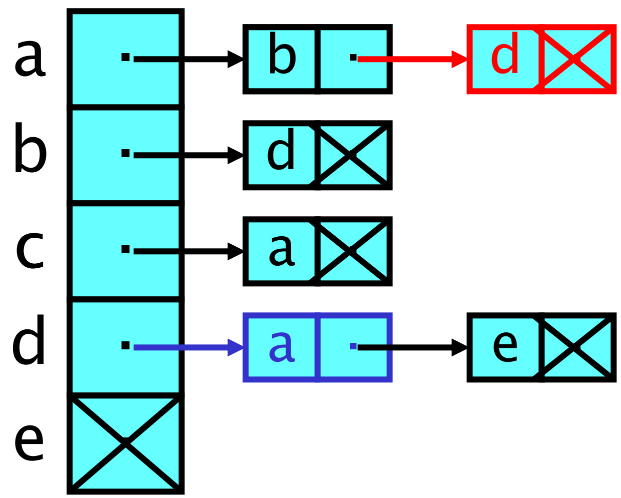


Graph Representation: Adjacency List

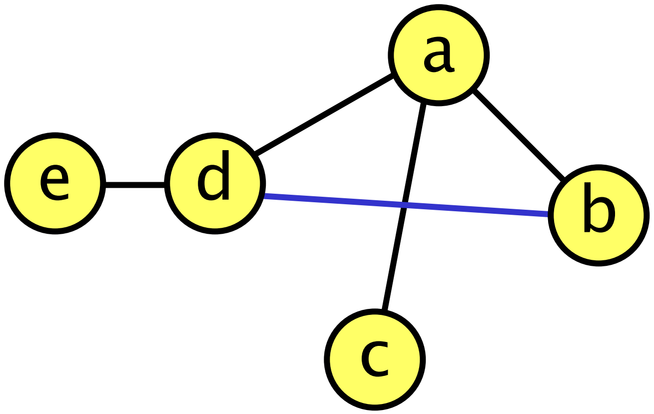




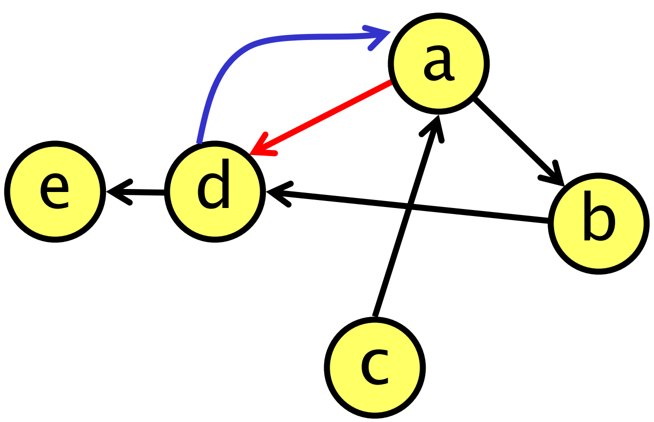




Graph Representation: Adjacency Matrix



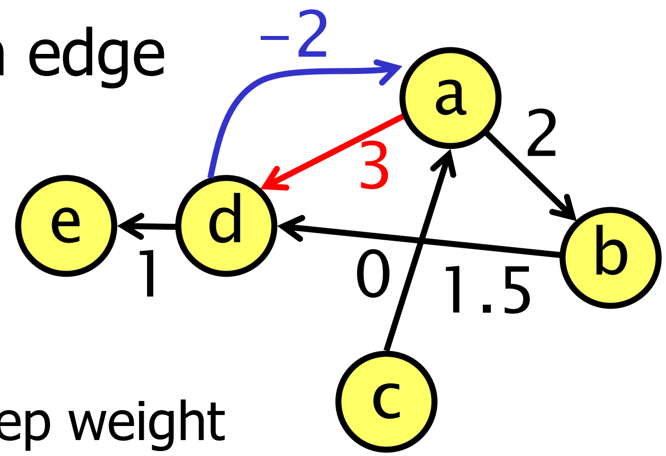
	a	b	c	d	e
a	0	1	1	1	0
b	1	0	0	1	0
c	1	0	0	0	0
d	1	1	0	0	1
e	0	0	0	1	0



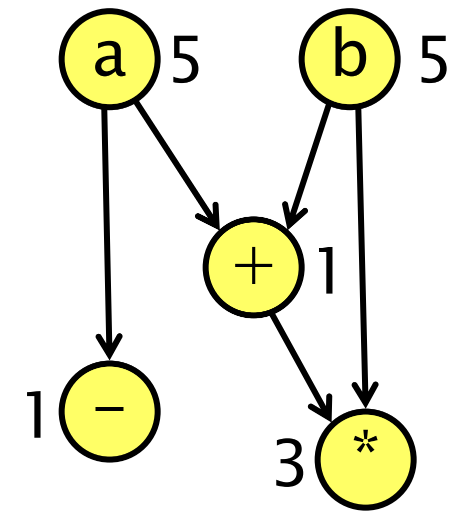
	a	b	c	d	e
a	0	1	0	1	0
b	0	0	0	1	0
c	1	0	0	0	0
d	1	0	0	0	1
e	0	0	0	0	0

Edge / Vertex Weights in Graphs

- Edge weights
  - Usually represent the "cost" of an edge
  - Examples:
    - o Distance between two cities
    - o Width of a data bus
  - Representation
    - o Adjacency matrix: instead of 0/1, keep weight
    - o Adjacency list: keep the weight in the linked list item

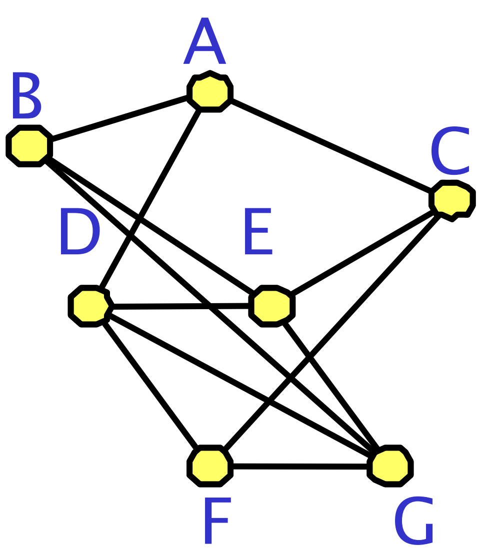


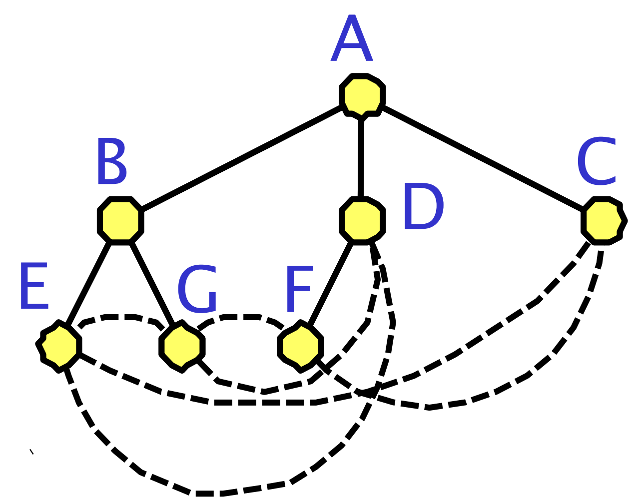
- Node weight
  - Usually used to enforce some "capacity" constraint
  - Examples:
    - o The size of gates in a circuit
    - o The delay of operations in a "data dependency graph"

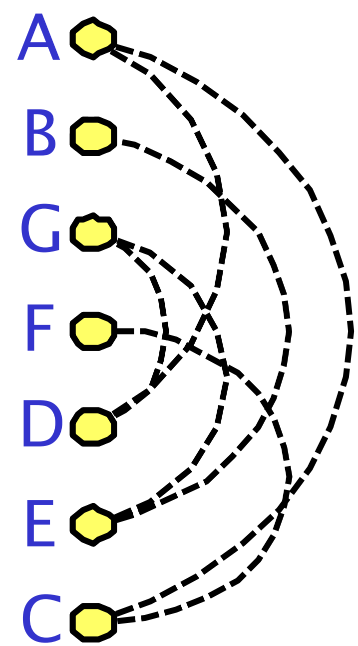


Graph Search Algorithms

- Purpose: to visit all the nodes
- Algorithms
  - Depth-first search
  - Breadth-first search
  - Topological
- Examples







[©Sherwani]

Depth-First Search Algorithm

```
struct vertex {
    ...
    int mark;
};
dfs ( v )
     v.mark ← 1
     print v
     for each (v, u) ∈ E
           if (u.mark != 1)    // not visited yet?
              dfs (u) // note the recursive call
// DFS goes "deep" into graph in contrast to BFS which
// "sweeps" the graph (mark all adjacent vertices first)
// Time complexity O(V+ E)
Algorithm DEPTH_FIRST_SEARCH ( V, E )
      for each v ∈ V
            v.marked ← 0    // not visited yet
      for each v ∈ V
            if (v.marked == 0)
               dfs (v)
```

Minimum Spanning Tree (MST)

- Tree (usually undirected):
  - Connected graph with no cycles
  - $|E| = |V| - 1$
- Spanning tree
  - Connected subgraph that covers all vertices
  - If the original graph not tree,
    graph has several spanning trees
- Minimum spanning tree
  - Spanning tree with minimum sum of edge weights (among all spanning trees)
  - Example: build a railway system to connect N cities, with the smallest total length of the railroad

Minimum Spanning Tree Algorithms

- Basic idea:
  - Start from a vertex (or edge), and expand the tree, avoiding loops (i.e., add a “safe” edge)
  - Pick the minimum weight edge at each step
- Known algorithms
  - Prim: start from a vertex, expand the connected tree
  - Kruskal: start with the min weight edge, add min weight edges while avoiding cycles (build a forest of small trees, merge them)

Prim's Algorithm for MST

- Data structure:
  - S — set of nodes added to the tree so far
  - S' — set of nodes not added to the tree yet
  - T — the edges of the MST built so far
  - $\lambda(w)$ — current length of the shortest edge (v, w) that connects w to the current tree
  - $\pi(w)$ — potential parent node of w in the final MST (current parent that connects w to the current tree)
- Time complexity is O(n2)

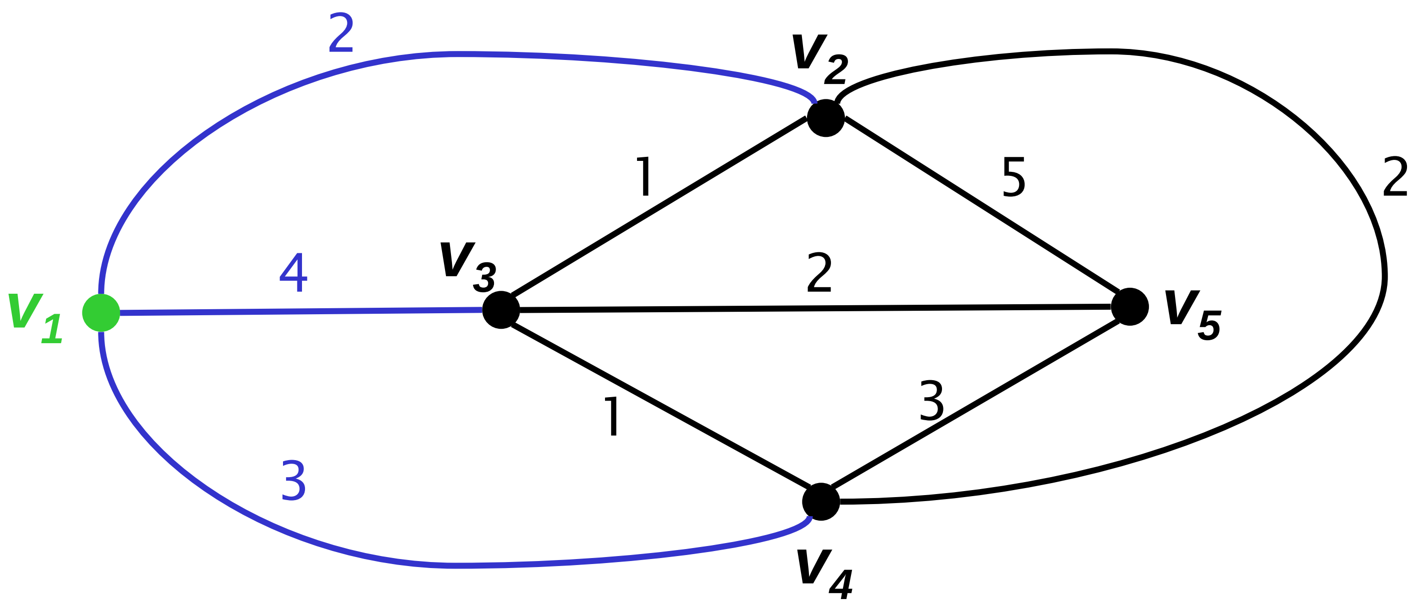
Prim's Algorithm

- Initialize S, S′ and T
  - o $S \leftarrow \{u_0\}$, $S' \leftarrow V - \{u_0\}$ // u_0 is any vertex
  - o $T \leftarrow \{\ \}$
  - o $\forall\ v \in S'$, $\lambda(v) \leftarrow \infty$
- Initialize λ and π for the vertices adjacent to u_0
  - o For each $v \in S'$ s.t. $(u_0,v) \in E$,
    - $\lambda(v) \leftarrow \omega\ \boldsymbol{((u_0,v))}$ // set edge weights
    - $\pi(v) \leftarrow u_0$ // set parent node
- While ($S' \mathrel{!}= \phi$)
  - o Find $u \in S'$, s.t. $\forall\ v \in S'$, $\lambda(u) \leq \lambda(v)$ // pick least cost edge
  - o $S \leftarrow S \cup \{u\}$, $S' \leftarrow S' - \{u\}$, $T \leftarrow T \cup \{\ (\pi(u), u)\ \}$ // update
  - o For each v s.t. $(u, v) \in E$, // set new parent node & edge weights
    - If $\lambda(v) > \omega((u,v))$ then

      $\lambda(v) \leftarrow \omega((u,v))$

      $\pi(v) \leftarrow u$

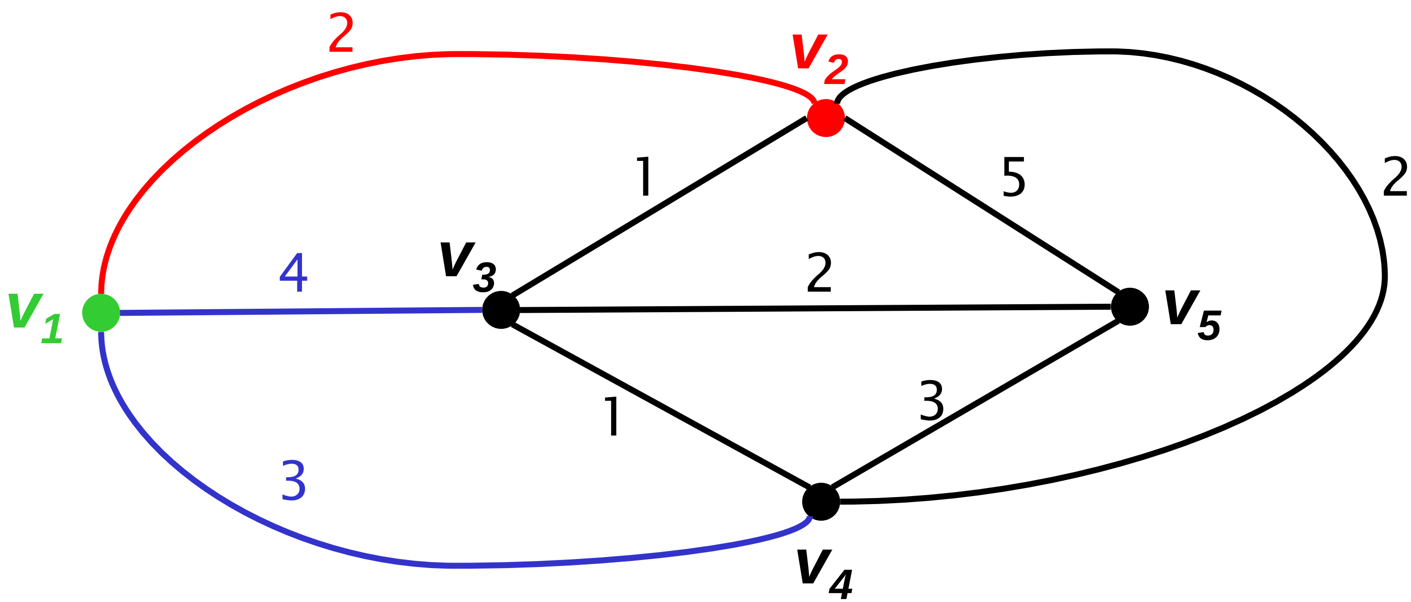
Prim's Algorithm Example



$S = \{v_1\}$

Node	v_1	v_2	v_3	v_4	v_5
λ	-	2	4	3	∞
π	-	v_1	v_1	v_1	-

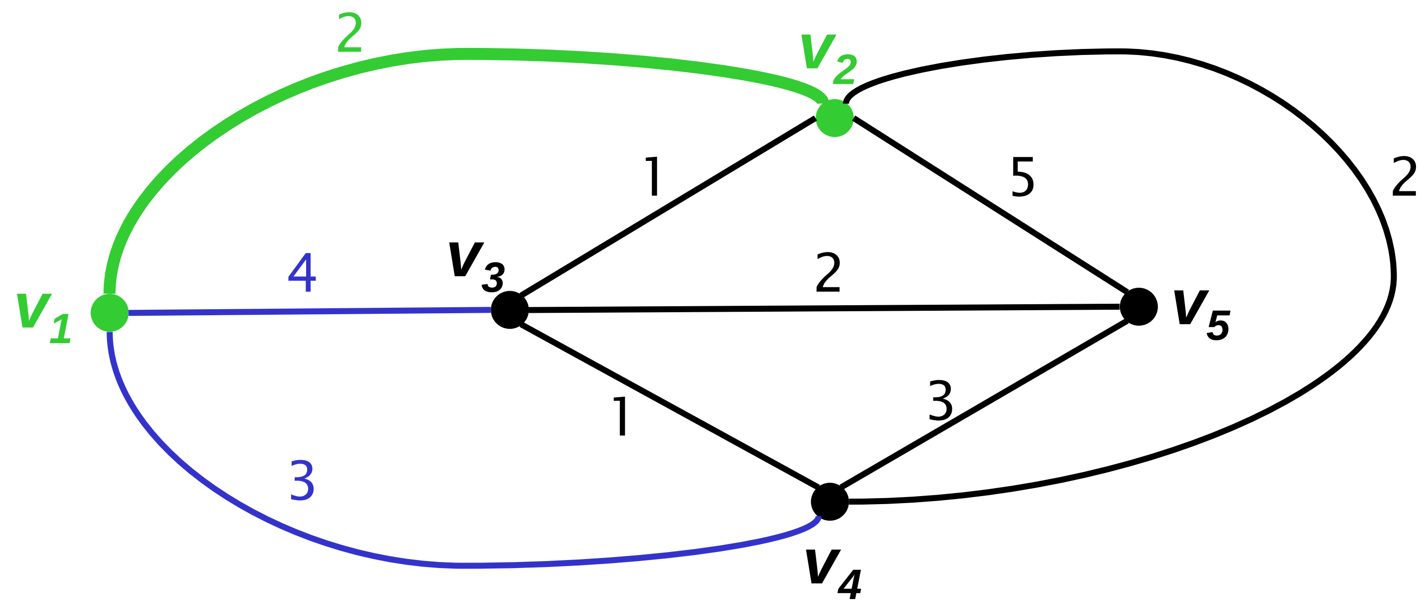
Prim's Algorithm Example



$S = \{v_1\}$

Node	v_1	v_2	v_3	v_4	v_5
λ	-	2	4	3	∞
π	-	v_1	v_1	v_1	-

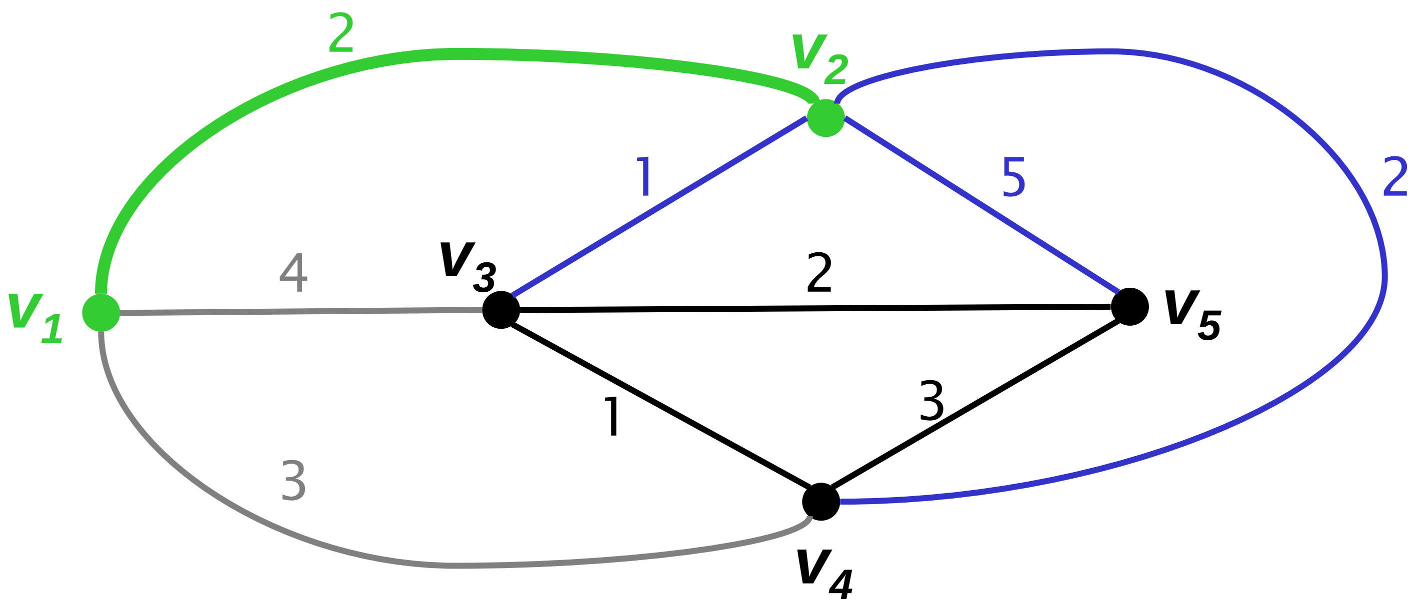
Prim's Algorithm Example



$S = \{v_1, v_2\}$

Node	v_1	v_2	v_3	v_4	v_5
λ	-	2	4	3	∞
π	-	v_1	v_1	v_1	-

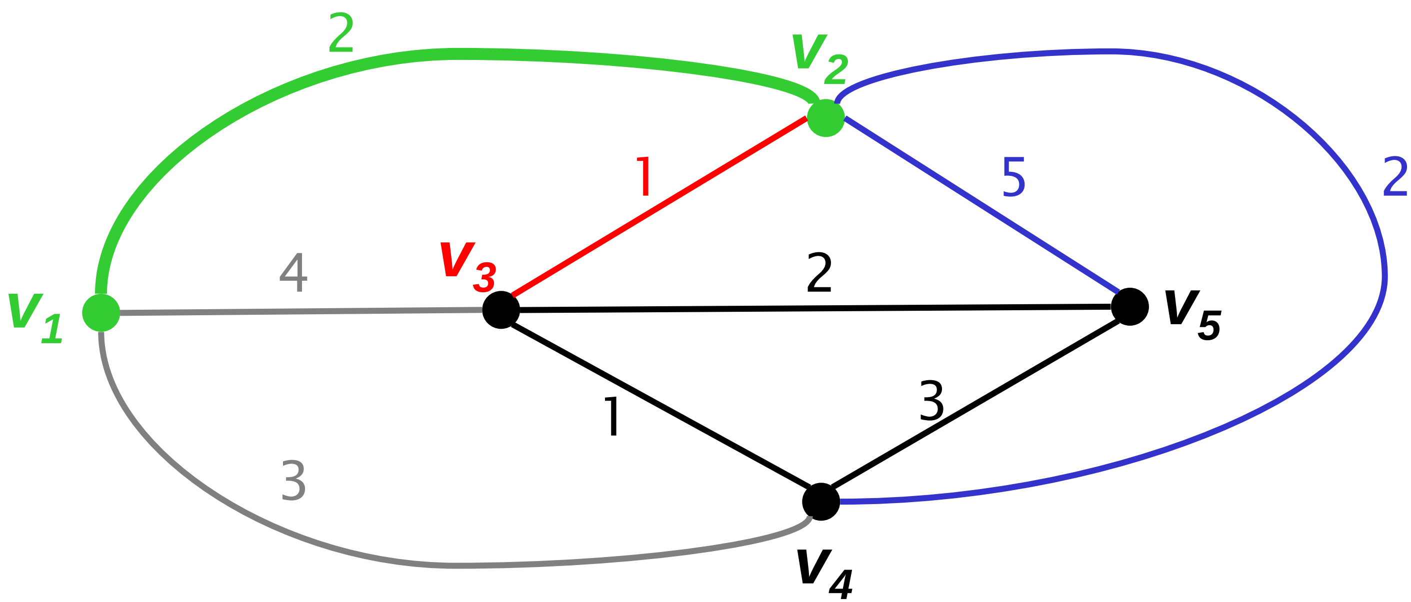
Prim's Algorithm Example



S = {v_1, v_2}

Node	v_1	v_2	v_3	v_4	v_5
λ	-	2	1	2	5
π	-	v_1	v_2	v_2	v_2

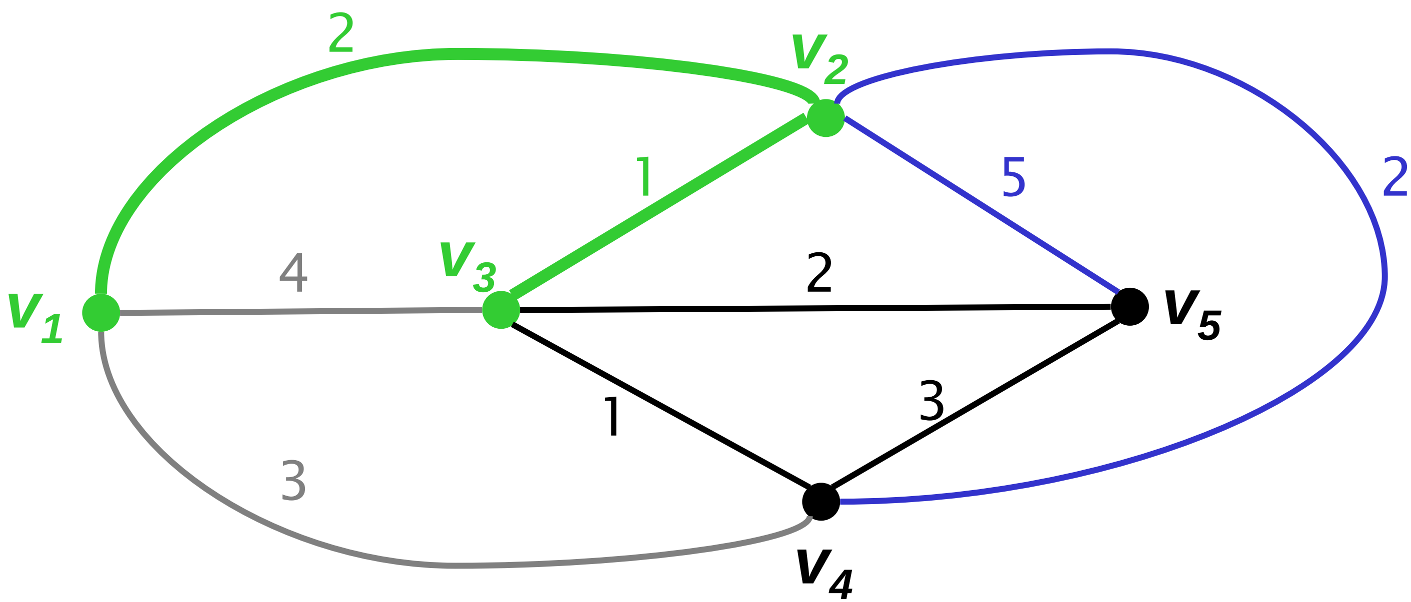
Prim's Algorithm Example



$S = \{v_1, v_2\}$

Node	v_1	v_2	v_3	v_4	v_5
λ	-	2	1	2	5
π	-	v_1	v_2	v_2	v_2

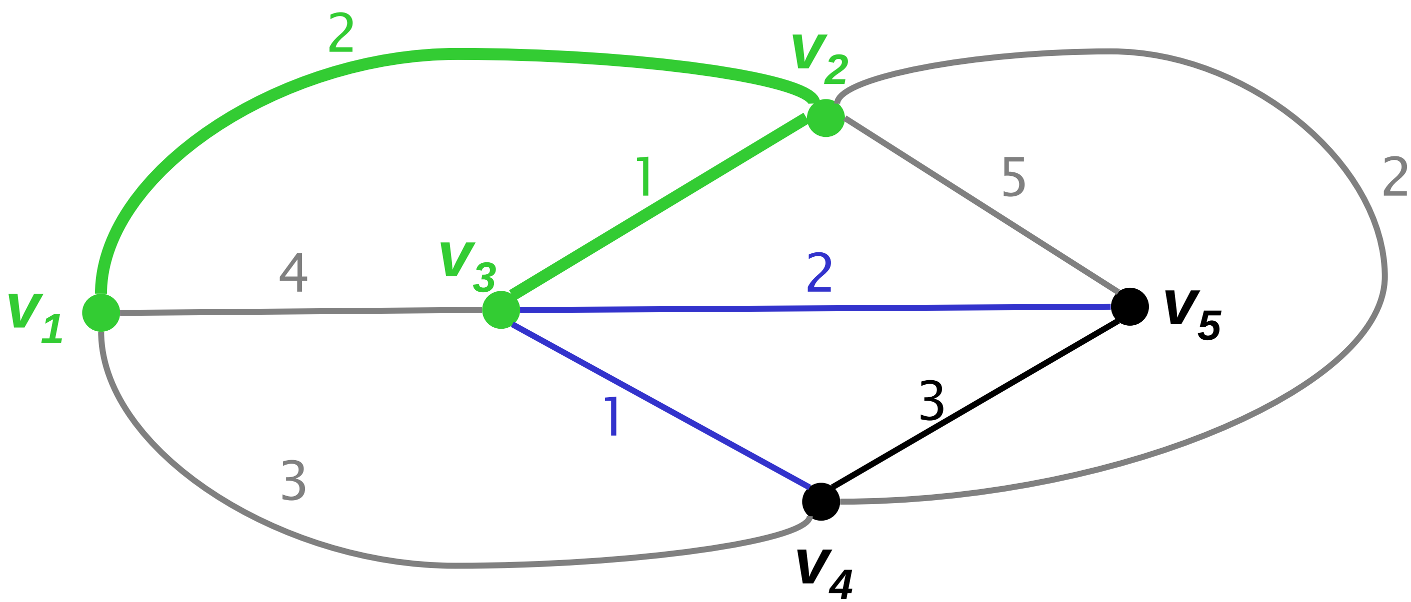
Prim's Algorithm Example



S = $\{v_1, v_2, v_3\}$

Node	v_1	v_2	v_3	v_4	v_5
λ	-	**2**	**1**	**2**	**5**
π	-	v_1	v_2	v_2	v_2

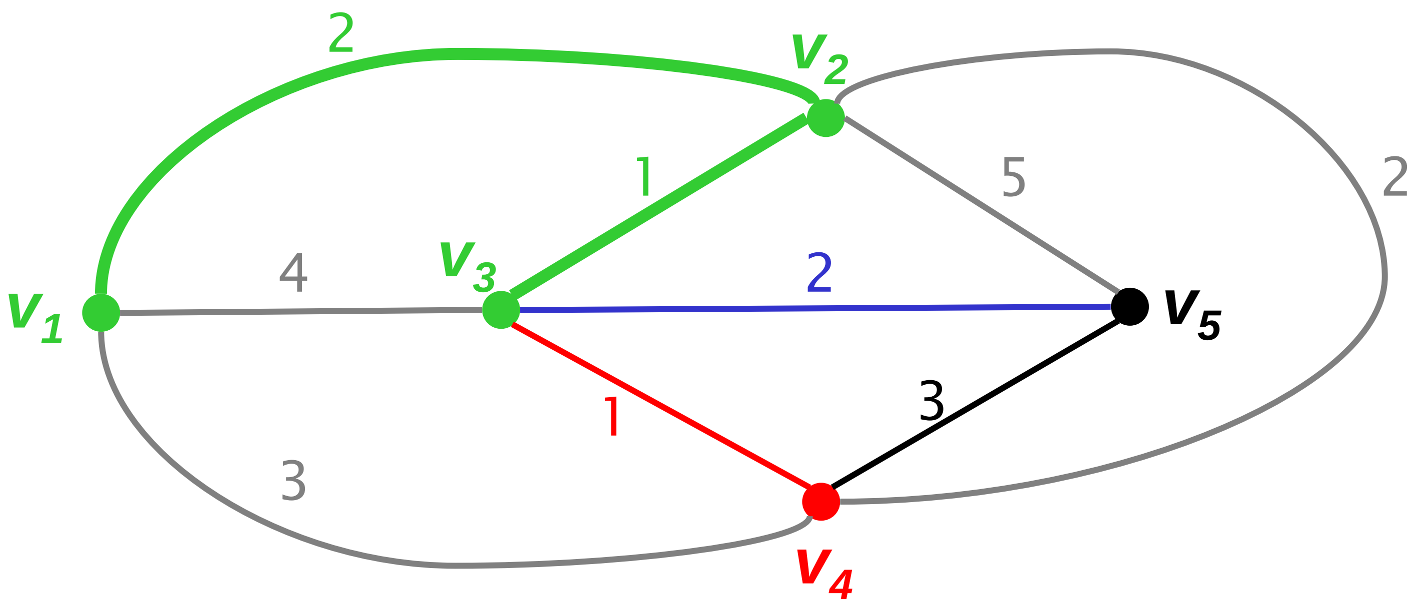
Prim's Algorithm Example



$S = \{v_1, v_2, v_3\}$

Node	v_1	v_2	v_3	v_4	v_5
λ	-	2	1	1	2
π	-	v_1	v_2	v_3	v_3

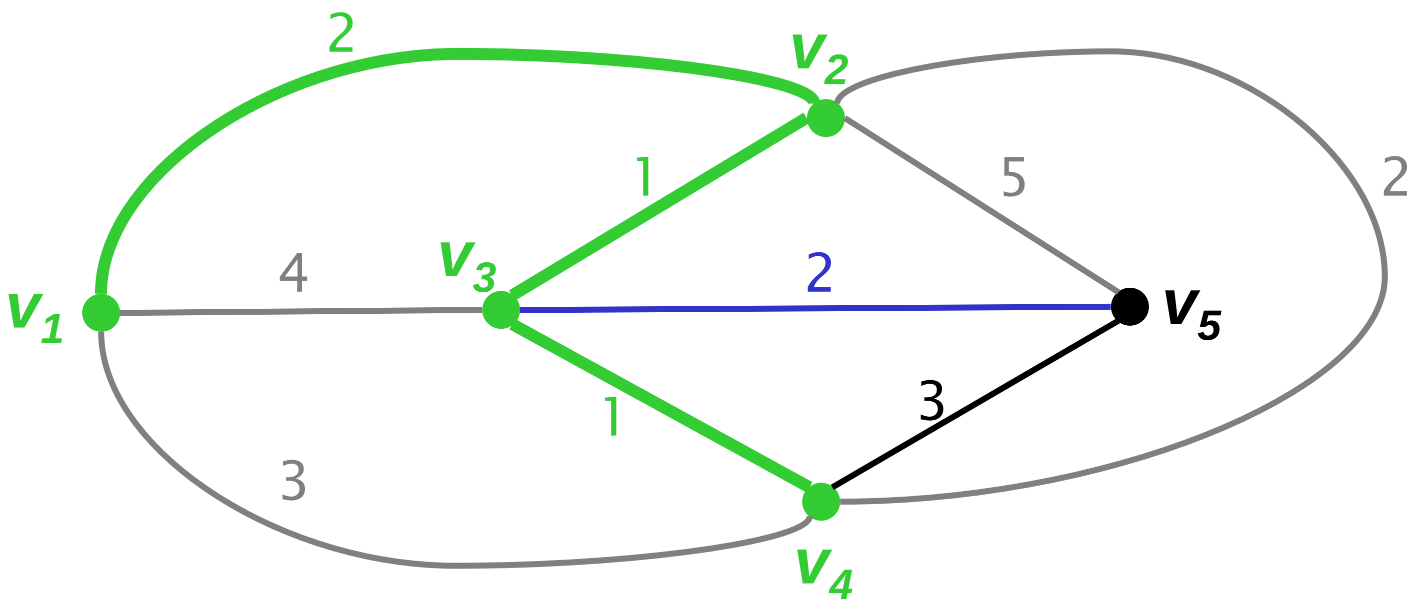
Prim's Algorithm Example



$S = \{v_1, v_2, v_3\}$

Node	v_1	v_2	v_3	v_4	v_5
λ	-	2	1	1	2
π	-	v_1	v_2	v_3	v_3

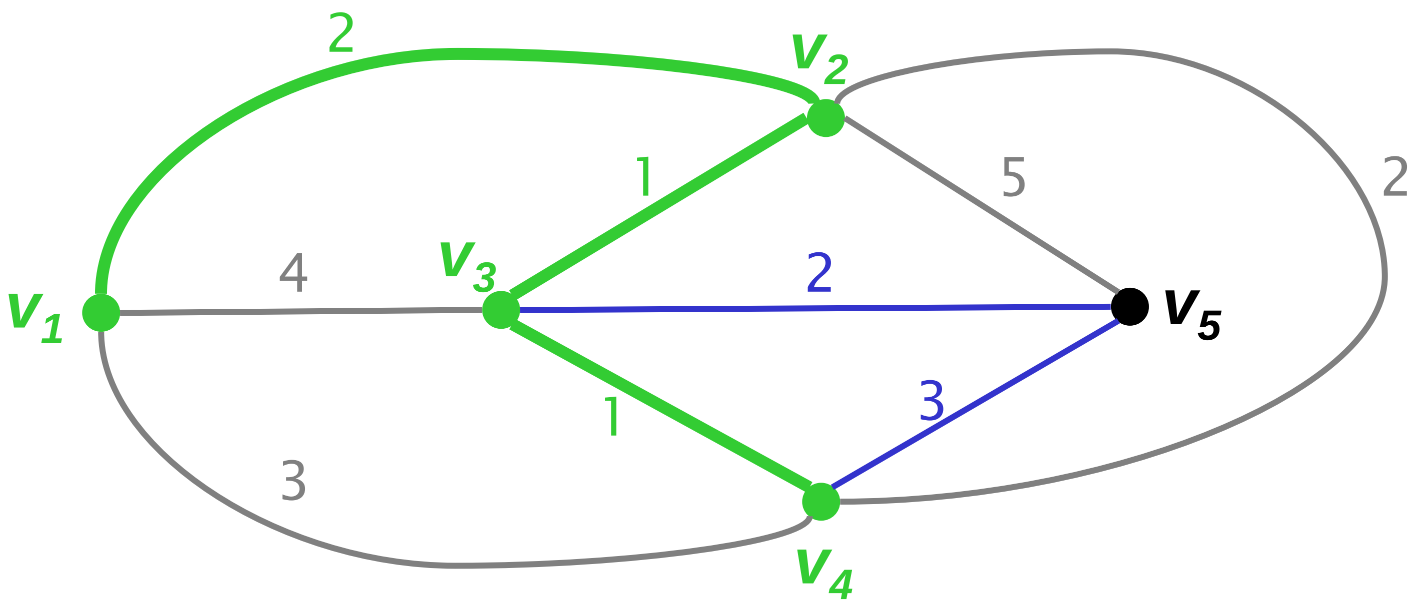
Prim's Algorithm Example



$S = \{v_1, v_2, v_3, v_4\}$

Node	v_1	v_2	v_3	v_4	v_5
λ	**-**	**2**	**1**	**1**	**2**
π	**-**	v_1	v_2	v_3	v_3

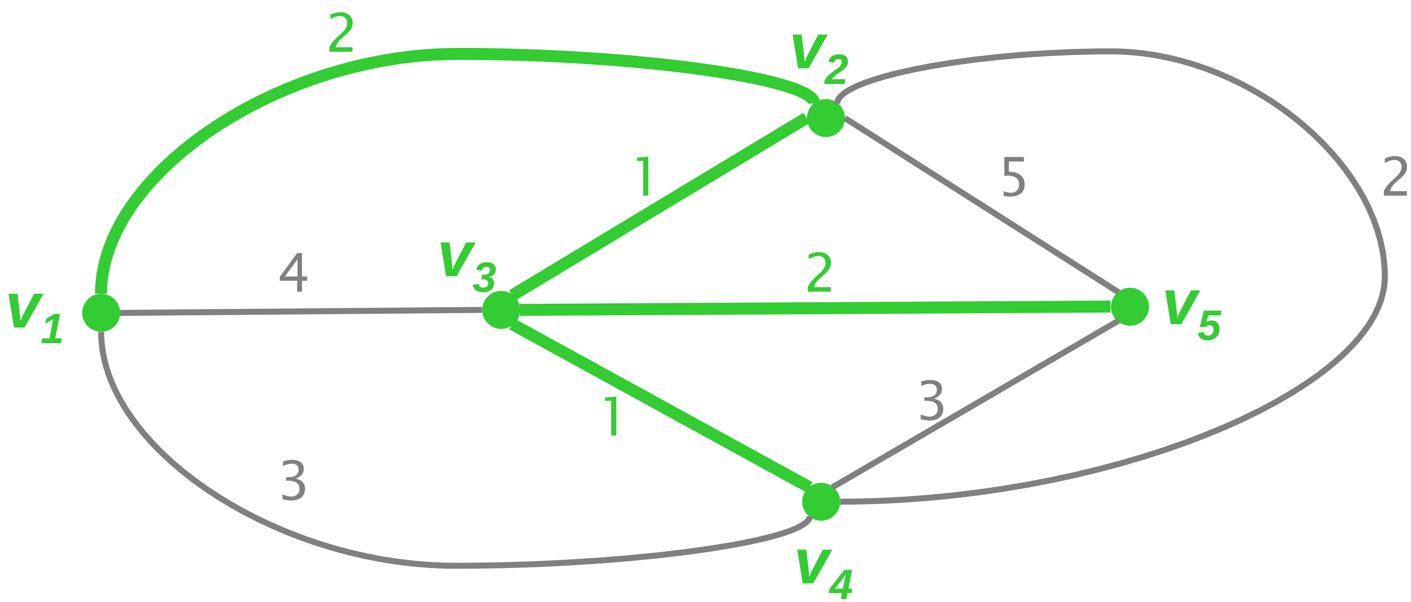
Prim's Algorithm Example



$S = \{v_1, v_2, v_3, v_4\}$

Node	v_1	v_2	v_3	v_4	v_5
λ	-	**2**	**1**	**1**	**2**
π	-	v_1	v_2	v_3	v_3

Prim's Algorithm Example



$S = \{v_1, v_2, v_3, v_4, v_5\}$

Node	v_1	v_2	v_3	v_4	v_5
λ	**-**	**2**	**1**	**1**	**2**
π	**-**	v_1	v_2	v_3	v_3

Other Graph Algorithms of Interest...

- Min-cut partitioning
- Graph coloring
- Maximum clique, independent set
- Min-cut algorithms
- Steiner tree
- Matching

- References for review
  - Any good Algorithms and Data Structures textbook
  - Wide variety of resources available on the web (search on Google, "tutorial on algorithms and data structures" or "specific topic")